Logic Meta Components as a Generic Component Model

Kris De Volder, Johan Fabry, Roel Wuyts

March 24, 2000

Abstract

Current component models depend on the modularization mechanism of the language in which they are defined. This significantly restricts what the functionality and contents of potential components can be.

We propose using a logic meta programming view on a software system. This allows not only component composition orthogonal to and independent of base language modularization mechanisms, but also the creation of generic components.

1 Introduction

Because current component models are basically defined on top of existing programming languages, they depend on the underlying modularization mechanism to define components and composition. This significantly restricts what the functionality and contents of potential components can be.

We propose a more general component framework based on a logic meta programming approach. In this approach, logic meta-programs are used to represent base-programs [DV98, DV99, DDVMW00]. This has as a consequence that the ability to compose meta-programs, automatically induces a composition mechanism at the base level.

This approach has two important advantages: First, by decoupling the component and composition mechanism from the base language, we can now define components and composition orthogonal to base language modularization mechanisms. Using AOP terminology, our components can 'cross-cut' base-level modules. Second, these components can be made to be more generic, by using the power of the logic language to script them together.

2 Defining Logic Meta Composition

The concept of logic meta-programming is straightforward: A base program is represented indirectly by means of a set of logic propositions. Since a logic program can be thought of as representing the set of logic propositions that can be proven from its facts and rules, these facts in turn can be thought of as indirectly representing a base-language program. The full power of the logic paradigm may thus be used to describe base-language programs indirectly.

In this paper we illustrate the use of logic meta programming to define generic components. We define logic meta composition in two steps: First, a representation of the base-system elements in the meta-system is defined. Second, a component and composition mechanism of meta-programs is defined. Composition of meta-programs which represent base programs, indirectly induces a mechanism to compose base-language programs.

2.1 First step: Defining the Representation

A base program is represented indirectly by means of a set of logic propositions. The relationship between the base program and its logic representation is made concrete under the

form of a code generator: a program that queries the logic data repository and outputs source code in the base language.

The mapping scheme between logic representation and base program may vary and determines not only the kind of information that is reified and accessible to meta programs, but also the granularity with which base program elements can be subdivided and separated into components.

For example, consider the mapping which represents Java class declarations by means of facts, which state that the class has certain methods, instance variables or constructors.

The presence of a variable declaration in a class is represented by a fact of the form:

```
var(?Class,?VarType,?VarName).
```

A method declaration is asserted by a fact of the following form:

```
method(?Class,?ReturnType,?MethodName,?ArgList,
   {...method body...}).
```

Below is an example Java class declaration and its corresponding representation as a set of propositions.

```
class Stack {                          | class(Stack).
  int pos = 0 ;                        | var(Stack,int,pos).
  Stack() {                            | method(Stack,Object,peek,[],
    contents = new Object[SIZE];}      |   {return...}).
  public Object peek ( )  {            | method(Stack,Object,pop,[],
    return contents[pos];  }           |   {return...}).
  public Object pop ( )  {             | ...
    return contents[--pos];  }
  ... }
```

2.2 Second step: Defining Components

The second step is defining composition mechanism at the meta-level. This boils down to defining a modularization mechanism for logic programs. Many such mechanisms have been proposed in the literature [Dav93].

For example, consider a Visitor [GHJV95] component: At the base level, the implementation of a Visitor is currently spread over different classes; a visitor and a number of visited classes. However, at the meta level, we can package all of the clauses describing the Visitor component in a separate 'meta-component'.

The following code is a possible implementation of such a visitor component. The code is separated in three parts: a `REQUIRES` part, which states the context dependencies of the component, a `PROVIDES` part, which specifies the interface of the component, and a `CLAUSES` part, which contains the code for the component.

```
COMPONENT VisitorComponent
REQUIRES
   class("Tree"), subclass("Node","Tree"), subclass("Leaf","Tree"),
   different("Leaf","Node").
PROVIDES
   class("Visitor").
   method("Visitor", "Object","visit",[["Tree","t"]]).
CLAUSES
   class("Visitor").
   method("Visitor","Object","visit",[["Tree","t"]],
     {return t.accept(this);}).
   method("Visitor","Object","visitNode",[ ["Node","n"] ],
```

```
    {...implementation body...}).
  method("Visitor","Object","visitLeaf",[ ["Leaf","l"] ],
    {...implementation body...}).

  abstractmethod("Tree","Object","accept",[ ["Visitor","v"] ]).
  method("Node","Object","accept",[ ["Visitor","v"] ],
    { return v.visitNode(this); }).
  method("Leaf","Object","accept",[ ["Visitor","v"] ],
    { return v.visitLeaf(this); }).
END VisitComponent.
```

Note that we explicitly used the logic notation, describing the components using facts in the `CLAUSES` section. It is also possible to use a more user-friendly, Java-like syntax, since it can be translated relatively easily to its logic representation. We explicitly use the logic notation to emphasize that the nature of a 'component' is a logic description of a chunk of base level code.

The set of clauses contained in the component description above corresponds to a number of structural elements of the base level program: a class definition and a number of method definitions in classes.

Composition of such components with others is achieved simply by making a union[1] of all sets of clauses. We specify the context dependencies explicitly as a logic expression that must be satisfied. For this example, our visitor component inserts methods into the class hierarchy of `Tree`, `Node` and `Leaf`, which must therefore exist, as is noted in the requires part of the component declaration.

Similarly, the contractual interface of the component is specified by logic expressions in the provides part. In this example, it consists of the `visit` method defined on `Visitor`.

3 Genericity

3.1 Decoupling the Visitor from the Visited Classes

While the visitor component in the example above is reusable to some extent, its reusability remains limited because it lacks genericity and is highly dependent of the hierarchy of visited classes. This limits its reusability to application contexts which explicitly use the exact same class hierarchy.

The reusability of the component can be enhanced relatively easily by treating the classes more anonymously, by referring to them through logic variables, as illustrated below

```
COMPONENT VisitorComponent
REQUIRES
   :- nodeclass(?Node),leafclass(?Leaf),treeclass(?Tree),
      subclass(?Node,?Tree), subclass(?Leaf,?Tree).
PROVIDES
   class("Visitor").
   method("Visitor","Object","visit",[[?Tree,"t"]]).
CLAUSES
   class("Visitor").
   method("Visitor","Object","visit",[[?Tree,"t"]],
     {return t.accept(this);}).
   method("Visitor","Object",visit<?Node>,[ [?Node,"n"] ],
     {...implementation body...}).
   method("Visitor","Object",visit<?Leaf>,[ [?Node,"n"] ],
```

[1] This suffices to illustrate and experiment with the idea, but to make this approach scale up, a form of encapsulation and perhaps even inheritance will also be necessary. Such mechanisms have been explored extensively in the literature [Dav93]

```
    {...implementation body...}).

  abstractmethod(?Tree,"Object","accept",[ ["Visitor","v"] ]).
  method(?Node,"Object","accept",[ ["Visitor","v"] ],
    { return v.visit<?Node>(this); }).
  method(?Leaf,"Object","accept",[ ["Visitor","v"] ],
    { return v.visit<?Leaf>(this); }).
END VisitComponent.
```

This more generic visit component now refers to its contextual dependencies indirectly through logic variables. Thus, the concrete names of the classes can be provided at composition time, which makes the visitor component more reusable. This is achieved through declaring the predicates `nodeclass`, `leafclass` and `treeclass` to indicate concrete classes in some other logic meta component.

3.2 A Fully 'Generic' Visitor

The previous visitor is still not fully generic because it still contains code which is dependent on the existing class structure. More specifically, the code of the `visit<?Node>` and `visit<?Leaf>`, depends on the structure of the tree, and the contents of the nodes and leaves.

Therefore we can make the component more generic, by abstracting out the body of the visit methods, thus delegating the responsibility of filling in an appropriate implementation to another component. This makes our visitor component completely independent of the visited class hierarchy, making the component truly generic.

```
COMPONENT VisitorComponent
REQUIRES
   treeclass(?Tree),nodeclass(?Node),subclass(?Node,?Tree),
   visitNodeBody(?Node,?nodeName,?visitNodeBody).
PROVIDES
   class("Visitor").
   method("Visitor","Object","visit",[[?Tree,"t"]]).
CLAUSES
   class("Visitor").
   method("Visitor","Object","visit",[[?Tree,"t"]],
     {return t.accept(this);}).
   method("Visitor","Object",visit<?Node>,[ [?Node,?nodeName] ],
     ?visitNodeBody).

   abstractmethod(?Tree,"Object","accept",[ ["Visitor","v"] ]).
   method(?Node,"Object","accept",[ ["Visitor","v"] ],
     { return v.visit<?Node>(this); }).
END VisitComponent.
```

Note that the code relevant to `?Leaf` has been removed. Actually the difference between nodes and leaves is only related to the implementation of the visit methods, so there is no longer a need to make a distinction. To use this component, the `?Node` variable will have to be bound to a number of classes of the hierarchy which has to be visited, without distinction between leaves and nodes. In the next subsection we will explain how these bindings are performed in the composition process.

3.3 Composition

So far we have talked only about a single component but not how it will actually be used in a composition with other components. In this section we will elaborate our example to

show how to actually compose it with other components. Let's start with a component that provides a concrete hierarchy of classes to be visited.

```
COMPONENT ParseTree
REQUIRES
   nothing
PROVIDES
   abstractclass("AbstractExpression").
   class("NumberExpression").
   extends("NumberExpression","AbstractExpression").
   class("AdditionExpression").
   extends("AdditionExpression","AbstractExpression").
   class("MultiplicationExpression").
   extends("AdditionExpression","AbstractExpression").
CLAUSES
   class("NumberExpression").
   var("NumberExpression","int","value").
   ...
END VisitComponent.
```

Composing the ParseTree component with the Visitor component should now be possible in order to obtain a ParseTree implementation with a visitor. This boils down to defining a new logic meta component in terms of the ParseTree and the Visitor. The composed component describes how the two should link up their requires and provides parts. For example, the following is a PrintVisitor for the ParseTree.

```
COMPOSITION ParseTreeWithPrintVisitor<"ParseTree","VisitorComponent">
   treeclass("AbstractExpression").

   nodeclass("NumberExpression").
   visitNodeBody("NumberExpression","num",{System.out.print(num.value);}).

   nodeclass("AdditionExpression").
   visitNodeBody("AdditionExpression","addExp",{...visit addition...}).

   nodeclass("MultiplicationExpression").
   visitNodeBody("MultiplicationExpression","mulExp",{...}).

END ParseTreeWithVisitor.
```

This component specifies how the classes provided by the ParseTree component map onto the required Tree and Node classes of the Visitor. Note that more than one class in the ParseTree implementation can play the part of a node, each with its own implementation for the body of the visit method.

4 Conclusion

The Visitor example explained in the previous section illustrates how defining composition of base programs indirectly as composition of meta-programs representing them, naturally leads to a flexible composition mechanism. The main advantage illustrated in this example is how logic meta composition, being defined independently of the base language, can naturally handle components which cross-cut the base-language's modularization mechanisms.

The example also shows that highly generic and reusable components can be constructed. The latter depends on the power of the meta-language, on the one hand, to both parameterize the component, and, on the other hand, to also serve as a scripting language to fill in the

parameters at composition time . The connection between one generic component and a concrete application context is described using a logic program.

In fact, the example barely scratches the surface in illustrating the potential of this approach. Indeed, the logic 'programs' used in the visitor example are trivially simplistic and do not even contain any real logic programs. None of the logic code given above contains any rules, only facts are used. There is a lot of unharvested potential in the ability to use arbitrary logic programs in any section of the component or composition declarations.

For example, using a simple logic program in the composition of the ParseTree and the visitor we may write the following to declare that all subclasses of `AbstractExpression` are visitable nodes.

```
treeclass("AbstractExpression").
nodeclass(?NodeClass) :- subclass(?NodeClass,"AbstractExpression").
```

Also in the provides and requires parts of a component it is useful to write rules. For example to have more sophisticated verification that a component is correctly used in a certain application context, a more sophisticated logic script could be written to verify this. Or, alternatively, the provided features of a component may vary depending on its application context and be deduced using a script.

Likewise, component implementations themselves may partly consist of code generated by a script. This allows them to, for example, adapt more flexibility to different application context, make them even more reusable or more efficient.

Another interesting research track is how to refactor existing (legacy) code. We can use a combination of this component mechanism with a logic meta programming approach which uses the logic language to explore and discover complex patterns in existing code [Wuyss, Wuy98]. The resulting information can, for example, be used by meta-components to drive code generation and refactoring.

References

[Dav93] A. Davison. A Survey of Logic Programming-based Object Oriented Languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Trends in Object-Based Concurrent Computing*, pages 42–106. MIT Press, Cambridge, MA, 1993.

[DDVMW00] Theo D'Hondt, Kris De Volder, Kim Mens, and Roel Wuyts. Co-evolution of object-oriented software design and implementation. In *To appear in proceedings of the international symposium on Software Architectures and Component Technology 2000.*, 2000.

[DV98] Kris De Volder. *Type-Oriented Logic Meta Programming.* PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory, June 1998.

[DV99] Kris De Volder. Aspect-oriented logic meta programming. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, volume 1616 of *Lecture Notes in Computer Science*, pages 250–272. Springer Verlag, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison Wesley, Reading, MA, 1995.

[Wuy98] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS USA'98*, 1998.

[Wuyss] Roel Wuyts. *Design as an explicit abstraction over implementation.* PhD thesis, Vrije Universiteit Brussel, work in progress.